G.O.A.T CPU Report

Tyler Jurczyk, Jay Nathan, Soumil Gupta

Introduction & Project Overview

Mission
This project involves designing an out-of-order RISC-V based processor using register renaming, alongside other key features to enhance performance.

Functionalities Implemented/Work Completed
This project key features like register renaming, reservation stations, and multiple execution units were integrated. Components like branch predictors and commit buffers were optimized to improve performance. The overall implementation supports fetch, decode, dispatch, issue, and commit stages, increasing the overall throughput. Additional features like prefetchers and advanced memory management were also added to increase performance in certain scenarios.

Division of Labor
Our group met multiple times a week, mostly in person at the DCL. Division of labor was mainly driven by individual preferences and comfort levels with coding.

Soumil:
Control Instructions
Implemented Fetch & Decode
Reservation Station, Functional Units, Debugging
Superscalar, Divider, Perceptron

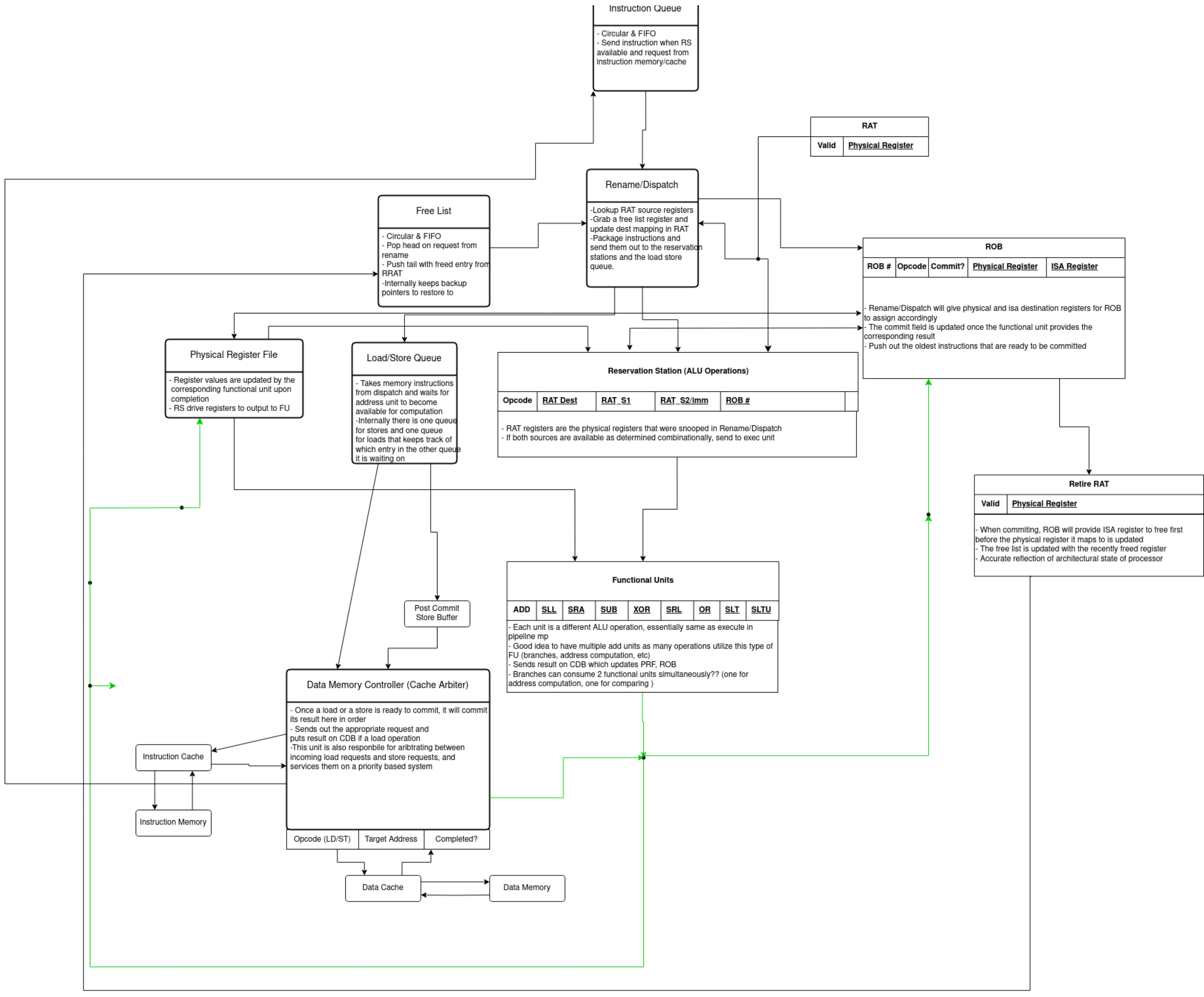
Tyler:
Memory Instructions
Cache Integration
Drew up datapath with planned features to implement
Circular Queue, Dispatcher, Free List,
Next Line Prefetcher, Load Store Queue,
Cache Arbiter, CDB,
Stride Prefetcher,
Post Commit Store Buffer,

J:
RAT, ROB
Cache Integration
Retire RAT, Top Level Design,
Physical Register File
Gshare

Testing Strategies
For testing strategies, we focused on verify base functionality of our processor through running directed local tests as well as coremark for verify the functionality of the entire processor. For the advanced features, we focused more on writing directed tests that adequately highlight the advanced feature's utility, while at the same time running extensive general tests to ensure that any old functionality wouldn't be broken. Additionally, these advanced features were manually verified in Verdi, as passing simulation wasn't a sufficient benchmark for correct performance of a feature. IPC was also checked, as even a moderate increase in IPC at least indicated the intentional functionality of the advanced feature. Finally, features were developed on separate branches and later merged into the main branch so that multiple features could be developed in parallel.

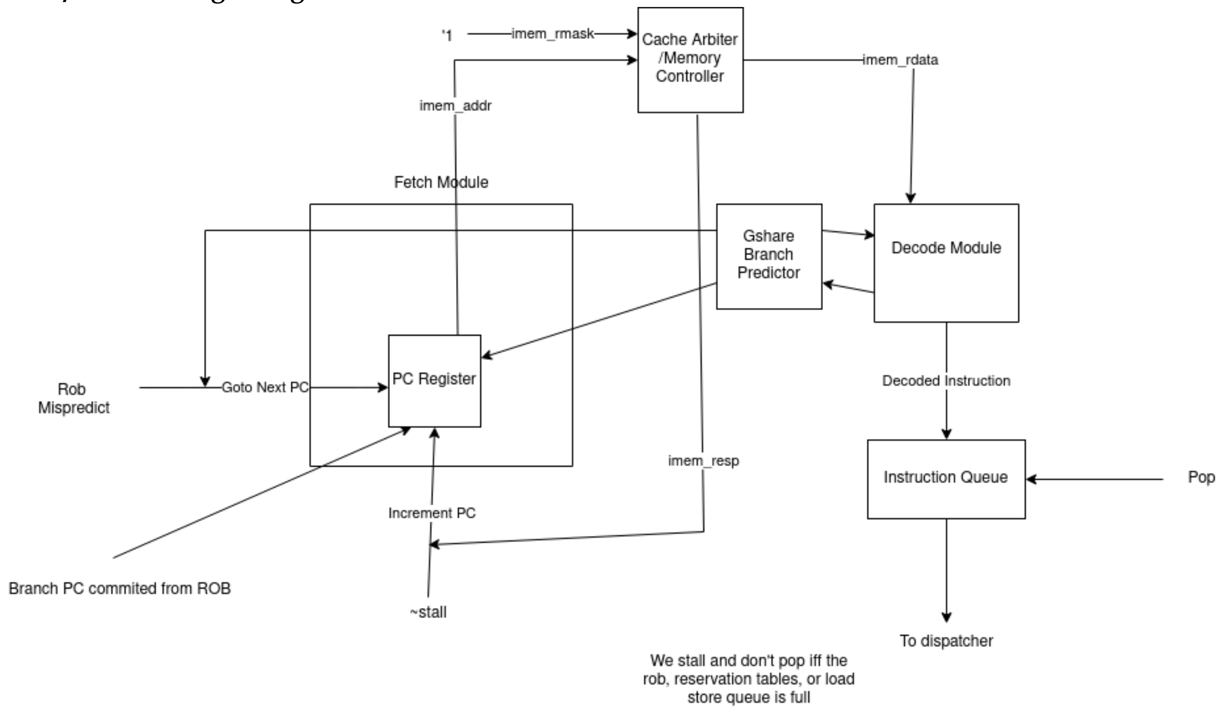
Design Description

Overview of full pipeline


Instruction Queue
- Circular & FIFO
- Send instruction when RS available and request from instruction memory/cache
RAT
Valid
Physical Register
Rename/Dispatch
-Lookup RAT source registers
-Grab a free list register and update dest mapping in RAT
-Package instructions and send them out to the reservation stations and the load store queue.
Free List
- Circular & FIFO
- Pop head on request from rename
- Push tail with freed entry from RRAT
-Internally keeps backup pointers to restore to
ROB
ROB #
Opcode
Commit?
Physical Register
ISA Register
- Rename/Dispatch will give physical and isa destination registers for ROB to assign accordingly
- The commit field is updated once the functional unit provides the corresponding result
- Push out the oldest instructions that are ready to be committed
Physical Register File
- Register values are updated by the corresponding functional unit upon completion
- RS drive registers to output to FU
Load/Store Queue
- Takes memory instructions from dispatch and waits for address unit to become available for computation
-Internally there is one queue for stores and one queue for loads that keeps track of which entry in the other queue it is waiting on
Reservation Station (ALU Operations)
Opcode
RAT Dest
RAT_S1
RAT_S2/imm
ROB #
- RAT registers are the physical registers that were snooped in Rename/Dispatch
- If both sources are available as determined combinationally, send to exec unit
Retire RAT
Valid
Physical Register
- When commiting, ROB will provide ISA register to free first before the physical register it maps to is updated
- The free list is updated with the recently freed register
- Accurate reflection of architectural state of processor
Functional Units
ADD
SLL
SRA
SUB
XOR
SRL
OR
SLT
SLTU
- Each unit is a different ALU operation, essentially same as execute in pipeline mp
- Good idea to have multiple add units as many operations utilize this type of FU (branches, address computation, etc)
- Sends result on CDB which updates PRF, ROB
- Branches can consume 2 functional units simultaneously?? (one for address computation, one for comparing)
Post Commit Store Buffer
Data Memory Controller (Cache Arbiter)
- Once a load or a store is ready to commit, it will commit its result here in order
- Sends out the appropriate request and puts result on CDB if a load operation
-This unit is also responbile for aribtrating between incoming load requests and store requests, and services them on a priority based system
Opcode (LD/ST)
Target Address
Completed?
Instruction Cache
Instruction Memory
Data Cache
Data Memory

Above is an overview of our Out of Order Processor with Explicit Register Renaming. It features a sequence of operations starting from an instruction queue to the dispatch of instructions to various execution units. Key components include a rename/dispatch unit which reduces false data dependencies by remapping registers, and a reservation station that queues instructions until all operands are available, directly feeding into instantiated functional units that run in parallel. Additionally, there is a reorder buffer to ensure a correct instruction commit order, and a post commit store buffer and cache arbiter for memory management. Finally RAT was utilized in order to keep track of speculative mappings from isa to physical registers, while a Retire RAT maintained the actual state of the processor.

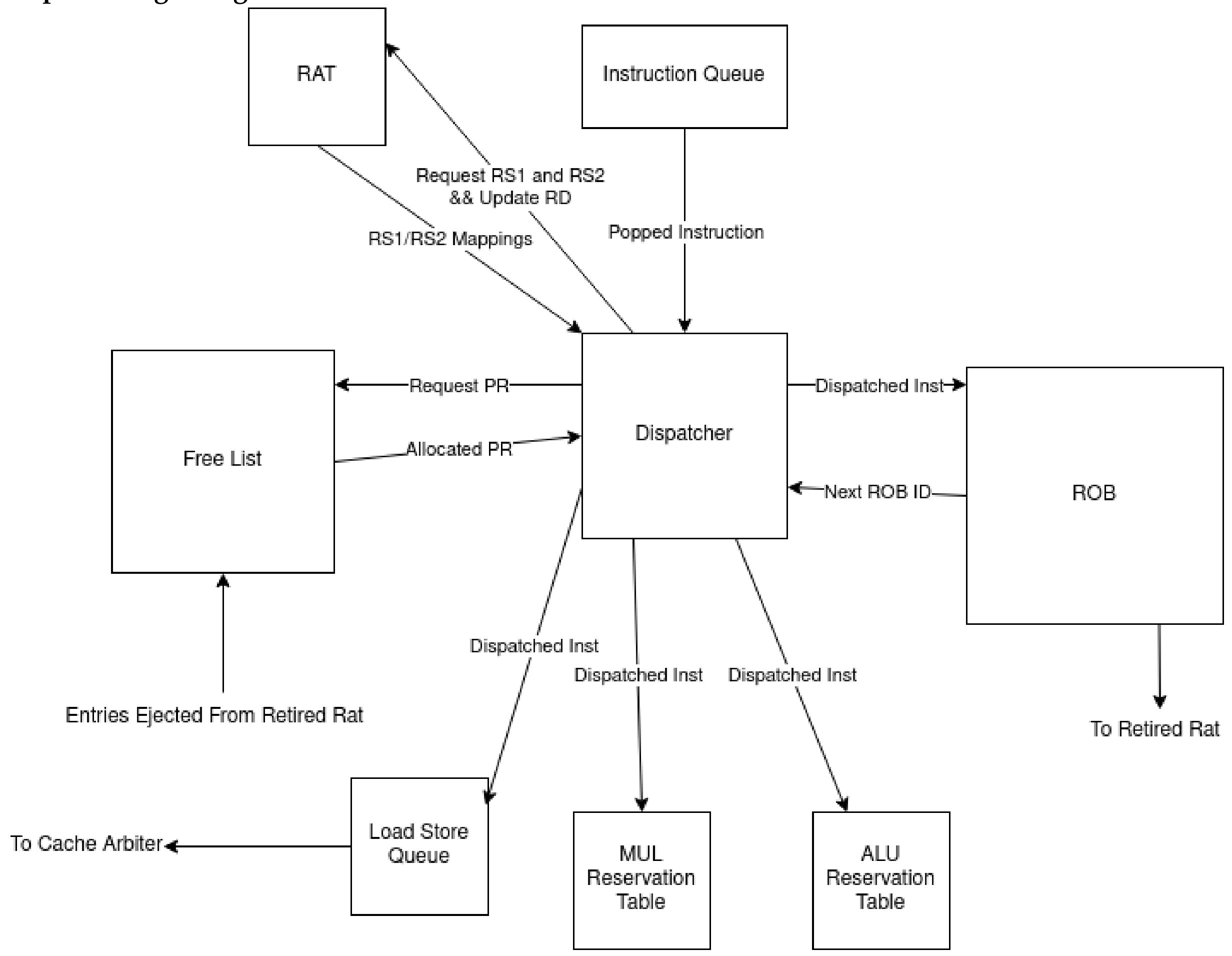
***Fetch/Decode Stage Diagram**



The Fetch/decode stage begins with the instruction fetching process, by taking instructions from memory. Instructions are fetched based on the address in the PC register, which is incremented only if there is not an active stall signal or a branch prediction updates the PC. Additionally, the Cache Arbiter acts as a controller to handle memory requests and returns the fetched instructions. At the same time, a Gshare Branch Predictor predicts the outcome of branches to update the PC, improving the fetch process for branch instructions.

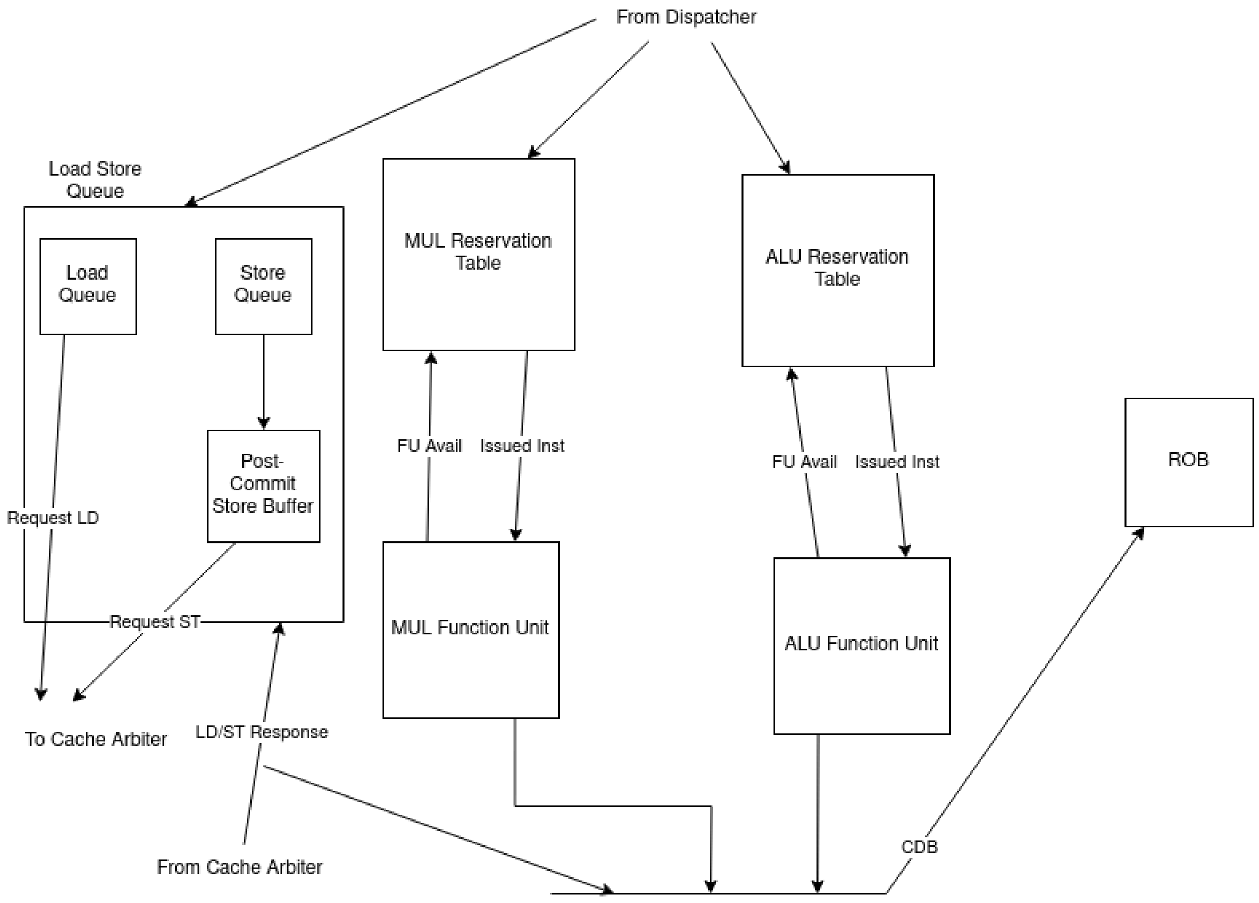
After fetching, the instructions pass into the decode process, where they are decoded into a format understandable by the processor. The decoded instructions are then entered into the Instruction Queue. This queue feeds into the dispatcher, which allocates the instructions into the respective execution units based on availability of the ROB, reservation tables, and load/store queue. This stage allows the instructions to be processed by managing the control and decoding logic, while maintaining a decent throughput and minimizing stalls within the pipeline.

***Dispatch Stage Diagram**



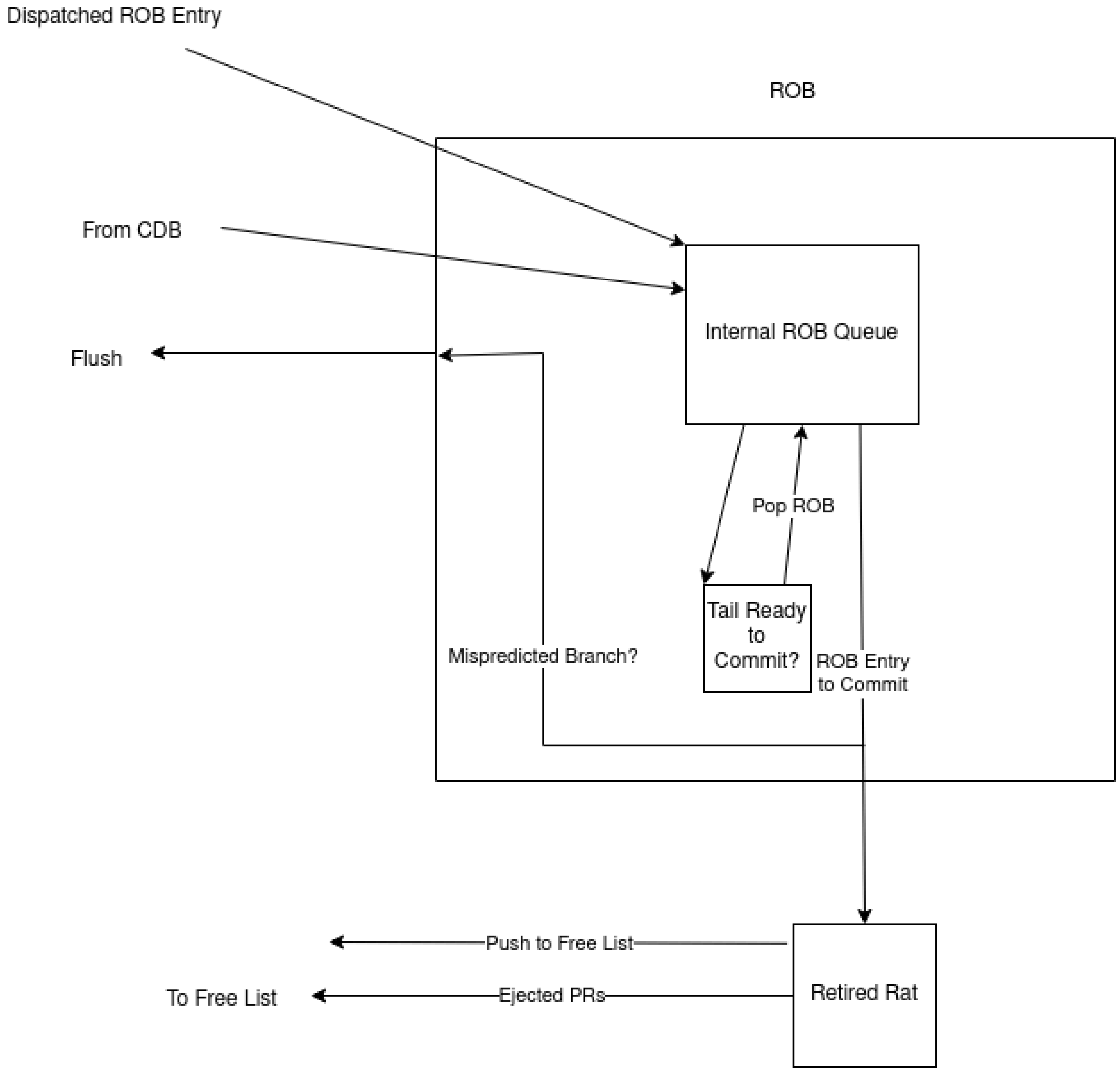
The dispatch stage in the processor manages the allocation of instructions from the instruction queue to the execution stages. It starts off with the RAT updating the physical register mappings for the destination registers and pulling the current mappings for the source registers. As instructions are fetched from the instruction queue, they are passed to the dispatcher to be conditionally processed. The dispatcher interacts with an instantiated freelist to allocate physical registers needed by destination registers for new instructions. It assigns these instructions to the appropriate reservation station based on the instruction type, ALU, MUL, or alternatively the Load/Store queue for memory instructions. The dispatcher also sends instructions into the ROB, making sure that each instruction is linked to the next available ROB ID.. This stage is important for keeping instruction flow and improving performance by minimizing execution delay and correctly dependency handling for the physical registers allocated to new instructions.

***Issue Stage Diagram**



The Issue stage in the processors pipeline manages the overall distribution of instructions between the different execution units. These instructions are sorted into MUL and ALU reservation tables based on their type, which then issues them to the corresponding MUL & ALU functional units as they become available. The Load/Store queue handles memory operations, by splitting tasks between separately instantiated load and store queues, in addition the Post-Commit Store Buffer which manages data writes. This stage oversees which execution units instructions are sent to, this allows the processor to effectively allocate internal resources by executing instructions out of order so long as the appropriate functional units and source registers are available.

***Commit Stage Diagram**



The Commit stage handles how instructions from the ROB are committed to change the architectural state of the processor. If a mispredicted branch is committed, then we flush the entire processor and update the program counter to fetch new instructions. Once instructions are committed from the ROB, given that they don't have a physical destination register of x0, they will eject the appropriate isa to physical register mapping the retired rat, and return the ejected physical register to the free list to be reused again for later instructions.

The ROB must for an instruction it is writing to first must identify whether the entry it is updating is for branch entry via the is_branch flag. If the entry is a branch, then another signal from the decode stage will state the branch prediction that is associated with that branch from the branch predictor. If there is a disparity between the prediction and the FU output, then flushing must commence. Without Early Branch Prediction, the entire pipeline and all bookkeeping is to be flushed.

All bookkeeping modules must be flushed. These modules include the ROB, Reservation Station, Instruction Queue. And the fetch module must be informed of the new PC to be fetching from that point onwards. Additionally, for register renaming, the RAT and Free Lists must be changed. The RRAT stores

the architectural state of the processor for before the Branch is to be committed. The RRAT for before any instructions from the branch and thus for flushing the processor, we want to not flush until the branch is the eldest instruction and ready to be committed in the ROB. The RRAT at this point will have a version of the architectural state of the processor before any of the mispredicted instructions from after the branch were issued, and thus the RAT will be replaced at this point. Additionally, a backup free list will also be saved upon every commit. The RAT and the free list will be replaced upon committing a mispredicted branch. Also, the fetcher will have to be updated with the new PC upon a mispredict.
Flushing just entails invalidating each entry in all queues and tables listed above.
The physical register file does not need to be updated because any destination registers with mappings and data from speculative instructions issued are no longer mapped to relevant instructions (will not get used).
Additionally, the CDB needs to get flushed in order to avoid writing something that is to be flushed.

Checkpoint 1 Milestones
For our first checkpoint, we primarily focused on how to design our processor in such a way as to allow the easy integration of advanced features early on. One of the major advance features we developed from the start was making the processor execute instruction in a 2-way superscalar fashion. This was done to catch as many bugs as early on as we could while developing this feature. Additionally we planned on implementing some advanced memory features such as cache line prefetching and a post commit store buffer so that we could improve the processors' performance once we switched to the banked memory model. As the for testing in the checkpoint, we simply attached out fetch/decode and circular queue directly to the magic memory model and manually verified that it was fetching instructions as we would expect

Checkpoint 2 Milestones
In the second checkpoint, we further built in the ability for our processor to scale up to superscalar, while at the same time doing testing on it in non-superscalar mode to first verify the base functionality. It was during this checkpoint that we also decided to split the reservation table into two, one dedicated for alu operations and the other dedicated for multiply operations. Other major parts of the out-of -order processor were developed during this checkpoint, with two of the most important ones being the rob and the physical register file. To properly handle dependencies and pass the dependency_test.s test case, we decided to implement dependency checking in the physical register file, and any time the physical register file was written to/read from, the dependencies on certain physical registers would be updated. Additionally, the reservation table kept a local copy of the dependencies so that it would know when to issue an instruction to the alu/multiply unit. To test functionality for this checkpoint, we used the given test cases along with hand written test cases to verify that dependency detection and resolution worked correctly, and that cdb would write correctly and update the physical register file appropriately.

Checkpoint 3 Milestones
For this final checkpoint, we implement the necessary branching and memory control instructions required to pass coremark. As a part of this, we started implementing some of the advanced features we planned on doing from early on, such as the next/stride cacheline prefetchers, and the post commit store buffer for storing writes prior to fully committing to memory. For branching, we utilized a static not-taken branch predictor, and anytime we had a branch commit from the rob to the retired rat, we would update the pc accordingly and flush all the instruction everywhere in the processor, including the rob, reservation tables, and the load store queues. For the load store queue, the loads and stores were split across two queues which would cross check each other before issuing instructions to memory so as to maintain their execution in order with respect to each other. One of the difficulties encountered during

this checkpoint was getting the flush timing correct, as there were certain edges that would result in a flush occurring while a store instruction was in the process of executing, which would cause the state of memory to change unintentionally. With regards to testing on this checkpoint, we focused on passing the base coremark after passing a few local directed tests for both the memory control instructions and branching separately. Once this baseline was achieved and we got coremark working, we continued to work on expanding the advanced features introduced in the first checkpoint, as well as implementing new ones such as a divider to help speed up division operations.

Superscalar (+15)

1. **Design**

   Fetch Stage

   For the fetch stage, in a 2-way superscalar processor, the fetch stage needs to build data for two instructions. The fetch stage will send a request for one PC, but both PCs will be sent over to the decoder stage. The imem_addr receives just one PC, which shall be the first of the two PCs. Additionally, the fetch stage receives two rob entries to commit, either of which could be containing a directive to jump/branch. What needs to be done is to maintain a pc_reg and and a pc_reg_2 that can be updated to pc_wdata and pc_wdata+4 when needed to. Additionally, if the instruction queue or the reservation table are full, we are to STALL. Also, if the branch predictor states to take a branch and any of the decoded instructions are a branch, or if any of the decoded instructions are a jump, you must then update your pc_reg and pc_reg_2 to that new PC.
   *Start hooking up with decode

   Dispatch Stage

   The instruction queue should receive and pop two instructions at the same time. However, the instruction queue must only pass instructions when the rob is not full and the topmost instructions in the queue's corresponding FU type and reservation station are not full.
   Upon a signal to pop from the queue, the instruction queue will pop two instructions. Two free registers must be popped off the free list. Then, the dispatcher must retrieve the RAT mapping for 2 sets of source registers and update the RAT mapping for 2 sets of destination registers. The dispatcher shall also enqueue the ROB ID dependencies to the physical register file and send out 2 entries to the ROB and reservation stations. The ROB always returns the next 2 available entries which can be utilized to update the physical register file instead of waiting for the ROB to enqueue said instructions.
   Two reservation stations are created, one for each type of instruction. The stations are parameterizable, where the decode stage assigned an FU type to each instruction, and then the dispatcher sends the same instructions to both reservations stations, but each reservation station will only enqueue the instructions depending on whether they are for the same FU they belong to.
   Because we have now doubled our I/O to N=2 wires, we now have to handle some additional conditions. We need a way to be able to read the topmost entry of the instruction queue for when we have a stall, as when the multiplication station is stalling, then we need to be able to pass in addition-based instructions through until we hit a multiplication instruction and the multiplication station is still full. To do this, a dummy instruction that will not impact the internal state of the processor needs to fill those wires because all wires need to be driven.
   Alternatively, if the the multiplication station is full, then you can check the instruction queue's N topmost instructions and see if they all depend on the FU reservation table, and then pop them both off.

   Reservation Station

   The reservation stations must be able to enqueue multiple instructions and issue multiple instructions. The reservation stations are already divided by the type of functional unit type. For issuing, the reservation table must be able to pop one OR two instructions at any given time, depending on the number of instructions the FU is able to entertain.

Prefetchers (+3 +4) (Next Line & Stride)

2. **Design**
   Two prefetchers were implemented to help improve memory access times. The first of these prefetchers was the next line prefetcher for the instruction cache, which would grab the next cacheline anytime the previous cacheline would have a request on it at a particular address. When a load or a store reached the cache, the cache would first check if a prefetch was in progress, and if so we would wait for the result to be returned. If not, we would send out a prefetch and proceed with processing the read request. Once in the idle state, the cache has logic to check if a prefetch we are waiting on is valid, and if so, modify the cache to contain this new cacheline.
   The second prefetcher that was implemented was the stride prefetcher for the data cache. This prefetcher sought to determine the stride of consecutive accesses, and if a pattern was found, prefetch the next line in the pattern. More generally, if caches that were N apart were being prefetched in X direct, get the next cacheline that was N away from the last accessed cacheline in the X direction. This prefetcher operated similarly to the next line prefetcher, in that it would prefetch the line, service the request, and load the prefetched cacheline once the cache controller returned to the idle state and the response was ready.
3. **Challenges**
   One of the challenges with implementing both of the prefetchers was ensuring that the prefetched cachelines would get correctly loaded, as the index needed to be changed to the prefetched address, and the appropriate way based on the plru would need to be modified/updated. To handle this, extra states were added to manage the loading of prefetched lines, which would give the SRAM adequate time to update the cacheline before servicing the next request.
4. **Testing**
   To test the functionality of the next line prefetcher, coremark was run and it was manually verified that the next cacheline would always be correctly fetched if requested on the current cacheline. For the stride prefetcher on the other hand, assembly was written specifically to access memory locations in stride as well as not in stride, which would help determine if the prefetcher was correctly detecting stride or not.
5. **Performance Analysis**
   Overall the performance increase appears negligible, with a slight increase for the next line prefetcher on coremark. The main reason for only a slight increase on the next line prefetcher comes down to the next line prefetcher being effectively negated on branches, and potentially slightly detrimental as extra states were added to the cache's fsm to properly handle the sending and receiving of prefetch requests.

Divider (+3)

The divider is the Synopsis IP implemented in account with their timing and utilizes the same infrastructure as the multiplier. The one interesting timing difference between the shift_add multiplier and the divider's I/O is the the fact that the divider gives an output for the input (or lack of) received at cycle 0, which means that the very first output has to be invalided. The IP was really interesting because the I/O is parametrically registered, and the divider had to be specified whether it was for unsigned or signed operations, which meant I had to instantiate two separate dividers with separate wrappers for each type of division operation. The decode stage was accordingly given further logic to determine the type of division the instruction will use, and another reservation station niche for dividers was connected to the rest of the CPU.

Testing was performed via running provided programs that had division operations. Through careful analysis of the timing diagrams provided in the IP documentation, there were only two timing bugs that had to be fixed.

In terms of performance, the divider plus the base CPU provided the most improvement to the processor as it decreases the number of instructions required to execute overall. The divider itself cut the performance from 4000 to 1300 in score.

Gshare (+4)

6. **Design**
   The gshare module integrates a Global History Register (GHR) and a Pattern History Table (PHT) to predict branch outcomes. It uses an XOR operation to blend the GHR with the branch address, indexing into the PHT of 2-bit saturating counters that adjust based on branch results. This setup lets us dynamically follow branching behaviors, and reduce stalls formed by mispredictions.
7. **Challenges**
   One challenge faced with implementing gshare was getting past the static prediction bias, where the PHT was initialized to strongly not taken (2'b00), introducing initial bias until the PHT adapts. To solve this problem the gshare was initialized to weakly not taken and dynamically adjusting the initial state based on early branch behavior.
   Another major issue was the overall size of the PHT and GHR, where initially the GHR was set to a length of 10 bits and the PHT to 1024 entries to match. This issue had an impact on branch prediction accuracy but had a massive area cost. To cut down on the area instantiated the GHR was set to 5-bits with the PHT having 32 entries.
8. **Testing**
   A series of branch-taken and not-taken signals were provided to test the branch predictor to verify if the GHR & PHT update correctly. For example, after a branch is taken, the GHR should shift left and insert a '1' at the least significant bit, and the corresponding PHT entry should increment if not at max saturation (2'b11).
   Additionally, tests to check that PHT entries saturate correctly at 2'b11 and 2'b00 for not taken without wrapping around. Another major test was to check the behavior of the first and last entries in the PHT and GHR sizes to check for overflow/underflow issues.
9. **Performance Analysis**
   As described above with the compromise of accuracy over area, we opted for a less accurate predictor and it shows not much of an improvement with the IPC seeing an increase of 5% .

Post-Commit Store Buffer with Write Coalescing (+6)

10. **Design**
    The idea behind the post-commit store buffer is to store requests in a buffer before writing them to memory at some point later on. The reason for this is by placing a committed store in a buffer, we won't have to immediately pay the performance penalty of waiting for a write to be serviced. Once a store is ready to commit as indicated by the rob and there are no pending loads, the store will be placed into the post-commit store buffer to be committed to memory later. In the case that the store buffer fills up, it will vacate all of its entries and write the data to cache, which may or may not need to first pull from main memory. If a load requests data that is completely present within the post commit store buffer, it will have its data forward from the buffer. In the case only part or none of the data is found in the buffer, it will first writeback all entries in the buffer corresponding to the cacheline that the load requests from, prior to servicing that load request. All data that is written from the post-commit store buffer to the cache is first coalesced into a cacheline before being written back.

11. **Challenges**
    One of the major challenges with implementing the post-commit store buffer is ensuring that

conflicting entries didn't overwrite in the wrong order. To ensure correctness, if an entry in the post commit store buffer had the same word-aligned address as an incoming store, the incoming store would modify the mask and the data appropriately to reflect the new data present at that address.

12. **Testing**

    Coremark was running to ensure the proper operation of the post commit store buffer, in part because coremark heavily utilizes loads and stores that are interdependent on each other. Additionally, it was manually verified that the post-commit store buffer was being updated and vacated correctly based on the conditions described in the Design section.

13. **Performance Analysis**

    Overall performance improved the IPC by 0.5% when running on coremark which is far less than expected, given that most of the time the cpu stalls is when waiting on memory operations to complete. One reason for this loss in performance is due to the aggressive writeback policy, whereby if the post commit store buffer fills up, it will vacate all its entries and write them back to main memory before proceeding with further memory requests. One way to optimize the case when the post commit store buffer is full is to just write back all the entries that makeup the most complete cacheline, rather than writing back all entries.

Perceptron with Overriding Branch Prediction

The perceptron was inspired by the original perceptron paper by DARPA. Once verified, it was connected with gshare and because perceptron takes multiple cycles, the perceptron has to flush and ignore some fetched instructions within the fetch stage.

The perceptron is a very powerful branch predictor. It is structured with a table of sets of weights, where each set of weights is in essence, one node in a neural net aka one perceptron. The program is broken down into segments of PCs, each corresponding to a perceptron in a perceptron "table" via an arbitrary hashing method. The perceptron weights are updated with the latest 16 (arbitrary number aligned with paper's findings) branch results, and the guess for a perceptron is calculated via those weights for a given perceptron hashed by the requesting branch's PC.

Testing was performed by comparing the IPC across branch-intensive programs, and putting a counter that could be viewed in Verdi that accumulated the number of mispredicts between the perceptron, and the Gshare, where it was found that the perceptron on average for a program, mispredicted about 45% less. The perceptron, while only truly capable of linear seperability, provided truly remarkable results. With the DNA test case, known for its quantity of branches, the IPC in comparison to the base processor went from 0.13 and 17.45 for the Gshare to a remarkable 0.41.

Conclusion, Sources, Closing Notes

Throughout the course of this project, we were able to develop an out of order processor that is based on the explicit register renaming model, alongside advanced features to help improve the performance of the processor. Throughout the course of implementing the advanced features, we often saw minimal improvement over the baseline, or at the very least, the performance gains weren't as much as initially expected. One reason for this is that many features introduced more overhead, with one example being the next line/stride prefetchers. This was because of additional states that were required for proper utilization of these prefetchers, however this overhead could be reduced by implementing other optimizations/features such a pipelined cache with prefetching built in. Some of the other features however did have noticeable performance improvement, one of those being the perceptron branch predictor, where the IPC improved by ~5% or more depending on the test cases that were run, and generally performed better compared to the baseline processor. When it came to implementing the post-commit store buffer, there was an IPC increase of ~1% for programs that had a lot of loads dependent on stores, because the processor could more often fetch the data it needed from the post-commit store buffer instead of having to go to the cache/main memory. If our processor were to be expanded on/reimplemented, it would have been beneficial to focus more on optimizing certain general features that produced benefits for a wide range of tests, such as the gshare branch predictor, instead of features that work only under specific test cases, such as the divider.